A Theoretical Foundation for Generation of Equivalent Transformation Rules (Program Transformation, Symbolic Computation and Algebraic Manipulation)

AUTHOR(S):

Akama, Kiyoshi; Koike, Hidekatsu; Miyamoto, Eiichi

数理解析研究所講究録
1125巻 2000年 44-58

A Theoretical Foundation for Generation of Equivalent Transformation Rules

Kiyoshi AKAMA† Hidekatsu KOIKE†† Eiichi MIYAMOTO††

†Center of Information and Multimedia Studies, Hokkaido Univ.
††Dept. of System and Information Eng., Hokkaido Univ.

Abstract

In the "Equivalent Transformation model" (ET model), computation is regarded as equivalent transformation (ET) of declarative descriptions, and programs consist of ET rules and control description. Programs may be improved by adoption of each new rule, and finally a correct and efficient program may be obtained.

A method for generating equivalent transformation rules from a given set of definite clauses is proposed. In this method, we generate new equivalent transformation rules by transforming meta-descriptions, each of which represents many problem descriptions. We also develop a theoretical foundation for the method based on the concept of equivalence of meta-descriptions.

1 Difficulty of Unfolding

1.1 Problem Solving by unfolding

Many problems can be solved by transforming the problem description equivalently into a simpler form.

For example, let P be a set consisting of the following definite clauses, where *app* means *append*.

$initial(X,Z) \leftarrow app(X,Y,Z).$
$app([A|X],Y,[A|Z]) \leftarrow app(X,Y,Z).$
$app([\,],Y,Y) \leftarrow .$

We consider two problems, (q_1,P) and (q_2,P), where

$q_1 = initial([1,2],[1,2,3,4]),$
$q_2 = initial(X,[1,2,3]).$

When given the first query q_1, we first make a set of definite clauses:

$P_1 = P \cup \{yes \leftarrow initial([1,2],[1,2,3,4])\}.$

Then, P_1 is transformed equivalently as follows (Only the *yes* clause in P_1 is changed.).

$yes \leftarrow initial([1,2],[1,2,3,4]).$
$yes \leftarrow app([1,2],Y,[1,2,3,4]).$
$yes \leftarrow app([2],Y,[2,3,4]).$
$yes \leftarrow app([\,],Y,[3,4]).$
$yes \leftarrow .$

From this transformation, the answer "yes" is obtained.

When given the second query

$q_2 = initial(X,[1,2,3]),$

we first make a new set of definite clauses:

$P_2 = P \cup \{ans(X) \leftarrow initial(X,[1,2,3])\}.$

Then, P_2 is transformed equivalently as follows (Only the *ans* clause in P_2 is changed.).

1: $ans(X) \leftarrow initial(X,[1,2,3]).$
2: $ans(X) \leftarrow app(X,Y,[1,2,3]).$
3: $ans([\,]) \leftarrow .$
$ans([1|W]) \leftarrow app(W,Y,[2,3]).$
4: $ans([\,]) \leftarrow .$
$ans([1]) \leftarrow .$

$ans([1,2|V]) \leftarrow app(V,Y,[3]).$

5: $ans([\,]) \leftarrow.$

$ans([1]) \leftarrow.$

$ans([1,2]) \leftarrow.$

$ans([1,2,3|U]) \leftarrow app(U,Y,[\,]).$

6: $ans([\,]) \leftarrow.$

$ans([1]) \leftarrow.$

$ans([1,2]) \leftarrow.$

$ans([1,2,3]) \leftarrow.$

Thus, the answer for the second problem is $X = [\,]$, $[1]$, $[1,2]$, or $[1,2,3]$. These two problems are easily solved by using unfolding in each step.

1.2 Meaning of Problems

A problem in this paper is determined by a pair (q, P) of an atom q and a set P of definite clauses. The correct answer for a problem (q, P) is the set of **all** ground atoms g such that g is an instance of q and g is a logical consequence of P.

For example, the correct answer for the first problem (q_1, P) is a singleton set

$\{initial([1,2],[1,2,3,4])\}$.

The correct answer for the second problem (q_2, P) is a set of four ground atoms

$\{\ initial([],[1,2,3]),$

$\ \ initial([1],[1,2,3]),$

$\ \ initial([1,2],[1,2,3]),$

$\ \ initial([1,2,3],[1,2,3])\}.$

Note that an element of these sets can **not** be regarded as the correct answer of problems.

1.3 Difficulty of Unfolding

Consider a predicate *primes*. $primes(X)$ is true when X is a list of the first n prime numbers ($n \geq 0$), i.e.,

$primes([\,])$	true
$primes([2])$	true
$primes([2,3])$	true
$primes([2,3,5])$	true
$primes([2,3,5,7])$	true
$primes([3,5])$	false
$primes([8])$	false

The predicate *primes* is defined by a set P_{primes} consisting of the following definite clauses:

$primes(Ps) \leftarrow gen(2,Ns), sift(Ns,Ps).$

$gen(N,[\,]) \leftarrow.$

$gen(N,[N|Ns]) \leftarrow add(N,1,N1), gen(N1,Ns).$

$sift([\,],[\,]) \leftarrow.$

$sift([X|Xs],[X|Ys]) \leftarrow filter(X,Xs,Zs),$
$\qquad sift(Zs,Ys).$

$filter(N,[\,],[\,]) \leftarrow.$

$filter(N,[X|Xs],Ys) \leftarrow divisible(N,X),$
$\qquad filter(N,Xs,Ys).$

$filter(N,[X|Xs],[X|Ys]) \leftarrow indivisible(N,X),$
$\qquad filter(N,Xs,Ys).$

$add(X,Y,Z) \leftarrow X + Y = Z.$

$divisible(N,X) \leftarrow N$ is a divisor of X.

$indivisible(N,X) \leftarrow N$ is not a divisor of X.

Obviously, the correct answer of a query $primes([X,Y,Z])$ is $\{primes([2,3,5])\}$. However, Prolog-like "left-to-right" control can **not** answer this problem correctly. Given the query $primes([X,Y,Z])$, Prolog first finds $primes([2,3,5])$. However, it goes into an infinite computation (The predicate *gen* generates natural numbers infinitely.) when we input ";" to find other solutions. Thus, Prolog can not know in a finite time that there is no solution other than $primes([2,3,5])$, which means that Prolog can **not** answer the query $primes([X,Y,Z])$ correctly.

2 Problem Solving by ET Rules

2.1 Overcoming the difficulty of Unfolding

Prolog-like "left-to-right" control can not answer the "primes" problem due to endless computation, even if the order of body atoms in P_{primes} is best arranged.

To overcome this difficulty, we adopt the Equivalent Transformation Paradigm [1] and use more than one ET rule instead of only one unfolding rule (See the ET rules in Section 3,

which can successfully solve the primes problem in a finite time.). ET rules are especially useful when infinite loops in computation are difficult to avoid by simple "left-to-right" control.

2.2 Equivalent Transformation Rules

Assume that two disjoint sets of predicates, R_P and R_R, are given. Predicates in R_P are used for atoms in declarative descriptions, and those in R_R are used for executable atoms by an evaluator.

$R_P = \{ans, primes, initial, app, max, \cdots\}$.
$R_R = \{=, >, \leq, \cdots\}$.

An ET rule on R_P and R_R is of the form ($n \geq 0$):

$\langle rulename \rangle$:
$H, \{Cs\} \rightarrow \{Es_1\}, Bs_1;$
$\rightarrow \{Es_2\}, Bs_2;$
$\cdots$
$\rightarrow \{Es_n\}, Bs_n.$

Here, $\langle rulename \rangle$ is the name of a rule, H is an R_P atom, Cs is a (possibly empty) sequence of R_R atoms, Es_i are (possibly empty) sequences of R_R atoms, and Bs_i are (possibly empty) sequences of R_P atoms. H is called the *head*, Cs the *applicability condition part*, each Es_i an *execution part*, and each pair of Es_i and Bs_i a *body* of the ET rule. The $\langle rulename \rangle$, the condition part, and each execution part are optional.

Assume that we are given an atom b in the body of a definite clause C. An ET rule of this form is applicable to the atom b iff there is a substitution θ such that

- the head H matches the atom b by a substitution θ (i.e., $H\theta = b$),
- each variable (in the ET-rule) that does not appear in the head H is replaced with a new variable by θ, and
- $Cs\theta$ is true (by the given evaluator).

When the rule is applied to a clause C at an atom b, C produces n or fewer clauses. Each new clause is obtained, after $Es_i\theta$ is executed successfully (by the given evaluator) with an answer substitution σ, by replacing $b\sigma$ in $C\sigma$ with $Bs_i\theta\sigma$.

When $n = 0$, the ET rule $(H, \{Cs\}.)$ is often written as

$H, \{Cs\} \rightarrow \langle false \rangle$.

If Es_i and Bs_i are empty, then $\langle true \rangle$ can be used in place of the empty i-th body $(\{Es_i\}, Bs_i)$.

2.3 Problem Solving by using ET Rules

To answer the first problem (q_1, P) mentioned above, we can use the following ET rules.

r1: $initial(X, Z) \rightarrow app(X, Y, Z)$.
r2: $app([A|X], Y, Z) \rightarrow \{Z = [A|W]\}, app(X, Y, W)$.
r3: $app([\,], Y, Z) \rightarrow \{Y = Z\}$.

The process of answering the query

$q_1 = initial([1,2],[1,2,3,4])$

is:

$yes \leftarrow initial([1,2],[1,2,3,4])$.
$yes \leftarrow app([1,2], Y, [1,2,3,4])$. by Rule r1
$yes \leftarrow app([2], Y, [2,3,4])$. by Rule r2
$yes \leftarrow app([\,], Y, [3,4])$. by Rule r2
$yes \leftarrow$. by Rule r3

On the other hand, the second problem can not be solved by these three rules (r1, r2, and r3):

$ans(X) \leftarrow initial(X, [1,2,3])$.
$ans(X) \leftarrow app(X, Y, [1,2,3])$. by Rule r1
—- Execution halts. —-

No rule can not be evoked to the last *ans* clause, and the transformation terminates since matching is used for the applicability test for rules (r1, r2, and r3) and there is no θ that satisfies one of

$initial(X, Z)\theta = app(X, Y, [1,2,3])$,
$app([A|X], Y, Z)\theta = app(X, Y, [1,2,3])$, and
$app([\,], Y, Z)\theta = app(X, Y, [1,2,3])$.

To obtain the answer to the second problem, the following rule is also introduced.

r4: $app(X, Y, Z) \rightarrow \{X = [\,], Y = Z\};$
$\rightarrow \{X = [A|V], Z = [A|W]\},$
$app(V, Y, W)$.

Now r1 and r4 solve the second problem (q_2, P).

1: $ans(X) \leftarrow initial(X, [1,2,3])$.
2: $ans(X) \leftarrow app(X, Y, [1,2,3])$. by r1
3: $ans([\,]) \leftarrow$.
$ans([1|V]) \leftarrow app(V, Y, [2,3])$. by r4
4: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2|W]) \leftarrow app(W, Y, [3])$. by r4
5: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2]) \leftarrow$.
$ans([1,2,3|U]) \leftarrow app(U, Y, [\,])$. by r4
6: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2]) \leftarrow$.
$ans([1,2,3]) \leftarrow$. by r4

2.4 Priorities for Rules

The two rules r1 and r4 can also solve the first problem:

$yes \leftarrow initial([1,2],[1,2,3,4])$.
$yes \leftarrow app([1,2], Y, [1,2,3,4])$. by r1
$yes \leftarrow app([2], Y, [2,3,4])$. by r4
$yes \leftarrow app([\,], Y, [3,4])$. by r4
$yes \leftarrow$. by r4

However, this computation is less efficient than the one by the rule set of r1, r2, and r3, for the following reasons:

- Given an *app* atom in the body of a definite clause C, Rule r4 can always fire and clause C is duplicated into two clauses, one of which is deleted by unifications ($X = [\,]$ and $X = [A|V]$).
- Rules r2 and r3 only examine a single matching to fire, while Rule r4 requires two unifications ($X = [\,]$ and $X = [A|V]$) at each step.

Thus, we give priority to each rule as follows:

1. Priority 1 to r1, r2, and r3.
2. Priority 2 to r4.

Rules (e.g., r4) with low priority can fire only when all the rules with high priority can not fire. Thus, the prioritized rule set

(Priority 1 = {r1,r2,r3},
Priority 2 = {r4})

can be used to solve two problems efficiently:

1. The first problem is solved by only the rules with Priority 1.
2. The second problem is solved by the rules with Priority 1 and 2 since an *app* atom that has a variable as the first argument can not be transformed by the rules with Priority 1.

2.5 More Efficient Rules

There exists a more efficient set of rules that can solve the first problem:

r5: $initial([\,], Z) \rightarrow \langle true \rangle$.
r6: $initial([A|X], [B|Z]) \rightarrow \{A = B\}$, $initial(X, Z)$.

These rules can solve the first problem as follows:

$yes \leftarrow initial([1,2],[1,2,3,4])$.
$yes \leftarrow initial([2],[2,3,4])$. by r6
$yes \leftarrow initial([\,],[3,4])$. by r6
$yes \leftarrow$. by r5

Similarly, the second problem can be solved by the rules:

r7: $initial(X, [\,]) \rightarrow \{X = [\,]\}$.
r8: $initial(X, [A|Z]) \rightarrow \{X = [\,]\}$;
$\rightarrow \{X = [A|W]\}$, $initial(W, Z)$.

By r7 and r8, the second problem is solved as follows.

1: $ans(X) \leftarrow initial(X, [1,2,3])$.
2: $ans([\,]) \leftarrow$.
$ans([1|W]) \leftarrow initial(W, [2,3])$. by r8
3: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2|V]) \leftarrow initial(V, [3])$. by r8
4: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2]) \leftarrow$.
$ans([1,2,3|U]) \leftarrow initial(U, [\,])$. by r8
5: $ans([\,]) \leftarrow$.
$ans([1]) \leftarrow$.
$ans([1,2]) \leftarrow$.
$ans([1,2,3]) \leftarrow$. by r7

We have the correct answer set:
$\{[\,],[1],[1,2],[1,2,3]\}$.

3 Rule Generation

3.1 Rule Generation Problem

In this paper, we propose a method for generating equivalent transformation rules from a given set of definite clauses (and a query). Since we have already constructed a system, called ETC, that translates a prioritized rule set into a C program, the main part of the program synthesis in the ET paradigm is to generate a prioritized set of ET rules.

For instance, given the first problem (q_1, P), our experimental program synthesis system can generate the following prioritized rule set:

R_1 = (Priority 1 = $\{r1, r2, r3\}$).

For the second problem (q_2, P), the system can generate

R_2 = (Priority 1 = $\{r1\}$,
Priority 2 = $\{r4\}$).

Based on the existence of the ETC system, (prioritized) rule sets are often regarded as programs.

The generated rules can solve the given problem. Furthermore, they can also solve other problems similar to the given one. For instance, R_1 can solve these problems (queries):

$initial([a,b,c],[a,b,c,d]) \rightarrow$ true.
$initial([8],[a,8,8,8]) \rightarrow$ false.
$initial([\,],[1,2,3]) \rightarrow$ true.
$initial([1,Y,Z],[1,2,3]) \rightarrow$ Y=2, Z=3.

3.2 Merging of ET rule sets

When we want to obtain a program that can answer both (q_1, P) and (q_2, P), one of the solution programs is

(Priority 1 = $\{r1, r2, r3\}$,
Priority 2 = $\{r4\}$),

which is obtained by merging the two rule sets (R_1 and R_2).

3.3 More Efficient Programs

Based on Section 2.5, we can obtain a more efficient program that can answer (at least) q_1 and q_2:

(Priority 1 = $\{r5, r6, r7\}$,
Priority 2 = $\{r8\}$).

This program can also be generated by our experimental program synthesis system and can answer, for instance,

$q_5 = initial([X,2|R],[3,Y,5,6])$

as follows.

1: $ans(X,R,Y) \leftarrow initial([X,2|R],[3,Y,5,6])$.
2: $ans(3,R,Y)$
$\leftarrow initial([2|R],[Y,5,6])$. by r6
3: $ans(3,R,2) \leftarrow initial(R,[5,6])$. by r6
4: $ans(3,[\,],2) \leftarrow$.
$ans(3,[5|W],2) \leftarrow initial(W,[6])$. by r8
5: $ans(3,[\,],2) \leftarrow$.
$ans(3,[5],2) \leftarrow$.
$ans(3,[5,6|V],2) \leftarrow initial(V,[\,])$. by r8
6: $ans(3,[\,],2) \leftarrow$.
$ans(3,[5],2) \leftarrow$.
$ans(3,[5,6],2) \leftarrow$. by r7

3.4 Examples of Rule Generation

3.4.1 max

Let P_{max} be a set of definite clauses that defines the maximum number in a number-list:

$max([A],A) \leftarrow$.
$max([A,B|X],Y) \leftarrow A > B, max([A|X],Y)$.
$max([A,B|X],Y) \leftarrow A \leq B, max([B|X],Y)$.

Given a query

$q = max([1,2,3,1],Y)$,

our system generates the following three rules:

$max([A],B) \rightarrow \{A = B\}$.
$max([A,B|X],Y),\{A > B\} \rightarrow max([A|X],Y)$.
$max([A,B|X],Y),\{A \leq B\} \rightarrow max([B|X],Y)$.

These rules can be used to compute the maximum number of numbers in a number-list.

3.4.2 primes

Given a query $primes([X,Y,Z])$ and a set of definite clauses, P_{primes} in Section 1.3, our sys-

tem generates the following rules:

$primes(Ps) \rightarrow gen(2, Ns), sift(Ns, Ps).$
$gen(N, [\,]) \rightarrow \langle true \rangle.$
$gen(N, [M|Ns]) \rightarrow \{\ N = M,$
$\quad add(N, 1, N1)\},$
$\quad gen(N1, Ns).$
$sift(Xs, [\,]) \rightarrow \{Xs = [\,]\}.$
$sift(Xs, [X|Ys]) \rightarrow \{Xs = [X|Rs]\},$
$\quad filter(X, Rs, Zs),$
$\quad sift(Zs, Ys).$
$filter(N, [X|Xs], Ys), \{divisible(N, X)\}$
$\quad \rightarrow filter(N, Xs, Ys).$
$filter(N, [X|Xs], Ys), \{indivisible(N, X)\}$
$\quad \rightarrow \{Ys = [X|Rs]\},$
$\quad filter(N, Xs, Rs).$
$filter(N, Xs, [X|Ys]), \{pvar(Xs)\}$
$\quad \rightarrow \{Xs = [X|Rs]\},$
$\quad filter(N, Xs, [X|Ys]).$
$filter(N, Xs, [\,]) \rightarrow \{\ Xs = [\,]\};$
$\quad \rightarrow \{\ Xs = [X|Rs],$
$\quad divisible(N, X)\},$
$\quad filter(N, Rs, [\,]).$

Only the last rule is given Priority 2 since it has two bodies. These rules can solve the primes problem and other similar problems such as $primes([2, 3, 5, 7])$ and $primes([2, 3, X, Y, 11])$.

3.4.3 common-subseq

A list X is a common subsequence of two lists Y and Z iff X is a subsequence of both Y and Z. A list X is a subsequence of a list L iff X is obtained by deleting some elements from L. Let $P_{common-subseq}$ be a set of definite clauses that defines the common subsequence relation:

$subseq([\,], X) \leftarrow.$
$subseq([A|X], [A|Y]) \leftarrow subseq(X, Y).$
$subseq([A|X], [B|Y]) \leftarrow subseq([A|X], Y).$
$common\_subseq(X, Y, Z) \leftarrow subseq(X, Y),$
$\quad subseq(X, Z).$

Given a query

$common\_subseq([1, 2], [1, 2, 3], [1, 2, 3, 4]),$

our system generates the following rules:

$common\_subseq([\,], Y, Z) \rightarrow \langle true \rangle.$
$common\_subseq([A|X], [\,], Z) \rightarrow \langle false \rangle.$
$common\_subseq([A|X], [B|Y], Z)$
$\quad \rightarrow equal(A, B),$
$\quad auto(X, Y, A, Z);$
$\quad \rightarrow common\_subseq([A|X], Y, Z).$
$auto(X, Y, A, [\,]) \rightarrow \langle false \rangle.$
$auto(X, Y, A, [B|Z])$
$\quad \rightarrow equal(A, B),$
$\quad common\_subseq(X, Y, Z);$
$\quad \rightarrow auto(X, Y, A, Z).$

The predicate *auto* is a newly generated predicate in the process of program synthesis by our system. These rules can check the *common_subseq* relation for all ground lists. Moreover, they can answer constraint satisfaction problems such as

$common\_subseq([1, X], [1, 2, 3], [0, 1, Y, 3]).$

4 Principle of Rule Generation

4.1 Program Synthesis and Rule Generation

The program synthesis discussed in this paper is defined as generation of a correct (and efficient) program to answer the query from a pair of a set of definite clauses and a query. In the "equivalent transformation (ET)" model, computation is regarded as equivalent transformation of declarative descriptions (e.g., a set of definite clauses) by a number of equivalent transformation rules. Therefore, generating ET rules that can transform a set of definite clauses equivalently is a foundation of the program synthesis in the ET paradigm. Due to space limitation, we discuss only the basic method of rule generation in this paper

4.2 Principle of Rule Generation

We adopt a new notion called meta-description, which consists of "meta-clauses." A **meta-description** is a representative expression for many declarative descriptions. A meta-clause is similar to a clause and consists of "meta-atoms." A **meta-clause** is a representative expression of many clauses. Similarly, a **meta-atom** is a representative expression of

many atoms. The purpose of introducing meta-descriptions is to reduce many ET sequences of descriptions into a single ET sequence of meta-descriptions.

Given a set of definite clauses and a meta-atom as an input pattern, we generate an ET rule as follows.

1. From the definite clauses, we prepare correct "meta-rules," which are used to transform meta-descriptions equivalently.
2. We determine the first meta-description from the given meta-atom.
3. We apply a meta-rule to the first meta-description to obtain the second meta-description and repeat application of meta-rules until we obtain the last meta-description.
4. We obtain an ET rule from the first and the last meta-descriptions.

In Sections 5, 6 and 7, we develop a general theory for rule generation, which will be used in Section 8 to generate ET rules for the term domain.

5 Specification and Program

5.1 Description and Its Meaning

In this paper, we use the same basic definitions as those used in the theory of logic programming [3], except for the word "program." We prefer the word **declarative description** (or simply **description**) to the word "program" to represent a set of definite clauses. Thus, a (declarative) description means a set of definite clauses in this paper.

Let $\mathcal{P}(\Sigma)$ be the set of all descriptions on an alphabet Σ, where Σ is a four-tuple of a constant set K, a function set F, a variable set V, and a predicate set R. Each element of K, F, V, and R is called a constant, a function, a variable, and a predicate, respectively. In the rest of this paper, Σ will not be mentioned for simplicity. For instance, $\mathcal{P}(\Sigma)$ is denoted by $\mathcal{P}$ and a description on Σ is called simply a description. The standard Prolog notation will be used for (declarative) descriptions. The meaning $\mathcal{M}(P)$ of a description P is a set of ground atoms that is determined by

$$\mathcal{M}(P) \stackrel{\text{def}}{=} T_P(\emptyset) \cup [T_P]^2(\emptyset) \cup [T_P]^3(\emptyset) \cup \cdots = \bigcup_{n=1}^{\infty} [T_P]^n(\emptyset),$$

where T_P is the "immediate consequence mapping," which is defined as follows. For any set x of ground atoms, $g \in T_P(x)$ iff there is a substitution θ and a definite clause C in P such that $C\theta$ is a ground clause, g is the head of $C\theta$, and all the body atoms of $C\theta$ belong to x.

5.2 Formalization of Problems

A problem is formalized as a pair (q, D) of an atom q and a declarative description D. A pair (q, D) determines a set by $rep(q) \cap \mathcal{M}(D)$, where $rep(q)$ is the set of all ground instances of q. Formalizing a problem in this paper is regarded as representing, by using a pair (q, D), the set to be found in the problem.

This formalization is consistent with the logical formalization in Section 1.2 since it is already proven that the set $rep(q) \cap \mathcal{M}(D)$ is equal to the set of all ground atoms g such that g is an instance of q and g is a logical consequence of P.

5.3 Specification

A specification is a pair (Q, D) of an atom set Q and a declarative description D. A specification (Q, D) determines the set $\{(q, D) \mid q \in Q\}$ of all problems that have to be solved.

For instance, let GL be the set of all ground lists and let

$$Q_1 = \{initial(x, y) \mid x \in GL, y \in GL\}.$$

Then, (Q_1, D_0) is a specification that requires judgement of whether x is a suffix of y for arbitrary ground lists x and y. Let

$$Q_2 = \{initial(x, y) \mid x \in V, y \in GL\}.$$

Then, (Q_2, D_0) is a specification that requires all suffixes x of an arbitrary ground list y to be found.

5.4 Problem Solving by Equivalent

Transformation

A specification (Q, D) can be regarded as a set of pairs of the form (q, D). A pair (q, D) formalizes a problem as finding a set $rep(q) \cap \mathcal{M}(D)$, which is called a **solution set**. Let $\hat{q}$ be an atom obtained by replacing the predicate r of q with a new predicate $\hat{r}$. Let ρ be a mapping such that, for any set X of ground atoms, $\rho(X)$ is the set of all ground atoms obtained by replacing the predicate $\hat{r}$ of $\hat{r}$-atoms in X with the old predicate r. It has been proven that

$$rep(q) \cap \mathcal{M}(D) = \rho(\mathcal{M}(D \cup \{\hat{q} \leftarrow q\})).$$

Let P_0 be $D \cup \{\hat{q} \leftarrow q\}$ and assume that $\mathcal{M}(P_0) = \mathcal{M}(P_n)$. Then,

$$rep(q) \cap \mathcal{M}(D) = \rho(\mathcal{M}(P_0)) = \rho(\mathcal{M}(P_n)).$$

Therefore, instead of computing $rep(q) \cap \mathcal{M}(D)$ directly, we can find the solution set by computing $\rho(\mathcal{M}(P_n))$. In the ET model, P_n is obtained from P_0 by repeated equivalent transformation.

$$\mathcal{M}(P_0) = \mathcal{M}(P_1) = \cdots = \mathcal{M}(P_n).$$

5.5 Equivalent Transformation Rules

A **rewriting rule** is a subset of $\mathcal{P} \times \mathcal{P}$. A description P_1 is transformed by a rewriting rule r into a description P_2, denoted by $P_1 \xrightarrow{r} P_2$, iff $(P_1, P_2) \in r$.

An **ET rule** r is a rewriting rule that satisfies

$$(P_1, P_2) \in r \rightarrow \mathcal{M}(P_1) = \mathcal{M}(P_2).$$

In order to emphasize that r satisfies this condition, an ET rule is also called a **correct** ET rule.

5.6 Correctness of Computation

The correctness of ET rules can be judged independently of other ET rules. As long as we use correct ET rules, the correctness of computation is guaranteed even though ET rules are applied in any order, which is formalized by the following theorem:

Theorem 1 *Let R be a set of ET rules. Let $r_1, r_2, r_3, \cdots, r_n$ be an arbitrary sequence of ET rules in R. If $P_0 \xrightarrow{r_1} P_1$, $P_1 \xrightarrow{r_2} P_2$, $\cdots$, $P_{n-1} \xrightarrow{r_n} P_n$, then $\mathcal{M}(P_0) = \mathcal{M}(P_n)$.* □

Proof. Let P_a and P_b be arbitrary declarative descriptions. Let $i = 1, 2, 3, \cdots, n$. Since $r_1, r_2, r_3, \cdots, r_n$ are equivalent transformation rules, if P_a is transformed by r_i into P_b, then $\mathcal{M}(P_a) = \mathcal{M}(P_b)$. Therefore,

$$\mathcal{M}(P_0) = \mathcal{M}(P_1) = \cdots = \mathcal{M}(P_n).$$

Hence, we have

$$\mathcal{M}(P_0) = \mathcal{M}(P_n).$$ □

5.7 Control by the Priority of Rules

The efficiency of computation depends on the selection of body atoms and ET rules. Therefore, control strategy is very important for constructing efficient programs.

One of the simplest ways to achieve efficient computation is to organize rules into mutually disjoint groups and put them in the order of preference for application. When equivalent transformation is controlled by the priority determined by the ordered rule sets $R_1, R_2, \cdots, R_n$, a rule r in the rule set R_i can be adopted only when all the rules in $R_1 \cup R_2 \cup \cdots \cup R_{i-1}$ are inapplicable but r is applicable. The union of all rule sets, $R_1, R_2, \cdots, R_n$, together with the priority of each rule is called a **prioritized rule set**.

6 Meta-description

6.1 Preliminaries

Let $\mathcal{A}$ be the set of all atoms on Σ and $\mathcal{S}$ the set of all substitutions on Σ. Let A be a subset of $\mathcal{A}$. A is **closed** iff $a\theta \in A$ for any atom $a \in A$ and any substitution $\theta \in \mathcal{S}$.

Let A_1 and A_2 be closed subsets of $\mathcal{A}$. A definite clause C is from A_1 to A_2 iff

1. all atoms in the body of C are elements of A_1 ($body(C) \subset A_1$), and
2. the head of C is an element of A_2 ($head(C) \in A_2$),

where $head(C)$ is the head of C and $body(C)$ is the set of all atoms in the body of C. The set

of all declarative descriptions that consist only of definite clauses from A_1 to A_2 is denoted by $Dscr(A_1, A_2)$.

6.2 Definition of Meta-Systems

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be mutually disjoint closed subsets of $\mathcal{A}$. A meta-system on $\mathcal{A}_1$ and $\mathcal{A}_2$ is a 5-tuple

$$\Delta = \langle \hat{\mathcal{A}}_1, \mathcal{A}_1, \Theta, \phi, \mathcal{B} \rangle,$$

of four sets $\hat{\mathcal{A}}_1$, $\mathcal{A}_1$, Θ, $\mathcal{B}$ and a mapping

$$\phi : \hat{\mathcal{A}}_1 \times \Theta \to \mathcal{A}_1,$$

where $\mathcal{B}$ is a subset of $\Theta \times \mathcal{A}_2 \times \mathcal{A}_1^*$ and $\mathcal{A}_1^*$ is the set of all finite subsets of $\mathcal{A}_1$. An element of the set $\hat{\mathcal{A}}_1$ is called a **meta-atom**.

6.3 Meta-clause and Meta-description

A **meta-clause** (on a meta-system Δ) is defined as an expression of the form ($m \geq 0$)

$$h \leftarrow \{\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n\},$$

where

1. the head is always h (called a dummy head), and
2. the body is a finite set of meta-atoms (on Δ).

Note that, in the formal theory, a body of a meta-clause is not a sequence of meta-atoms but a finite set of meta-atoms. However, in the case of informal discussion, this meta-clause is often written as

$$h \leftarrow \hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n.$$

This notation is also applied to the case of usual definite clauses.

A **meta-description** (on Δ) is a set of meta-clauses (on Δ). The set of all meta-descriptions on a meta-system Δ is denoted by $MD(\Delta)$.

6.4 Instantiation of Meta-clauses

Assume that $\beta = (\theta, H, B)$ in $\mathcal{B}$ is given. For an arbitrary meta-clause

$$\hat{C} = (h \leftarrow \{\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n\}),$$

we define $\psi(\hat{C}, \beta)$ as a clause

$$H \leftarrow \{\phi(\hat{B}_1, \theta), \phi(\hat{B}_2, \theta), \cdots, \phi(\hat{B}_n, \theta)\} \cup B.$$

In other words, $\psi(\hat{C}, \beta)$ is constructed from $\hat{C}$ by the following steps:

1. The body of $\hat{C}$, i.e., $\{\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n\}$, is "instantiated" by θ into a set S of atoms on $\mathcal{A}_1$, i.e.,
   $$S = \{\phi(\hat{B}_1, \theta), \phi(\hat{B}_2, \theta), \cdots, \phi(\hat{B}_n, \theta)\}.$$
2. The head of $\psi(\hat{C}, \beta)$ is H in β.
3. The body of $\psi(\hat{C}, \beta)$ is the union of the instantiated set S and B in β.

Since

1. H is an element of $\mathcal{A}_2$,
2. B is a finite subset of $\mathcal{A}_1$, and
3. $\phi(\hat{B}_i, \theta)$ is an element of $\mathcal{A}_1$,

$\psi(\hat{C}, \beta)$ belongs to $Dscr(\mathcal{A}_1, \mathcal{A}_2)$, i.e., $\psi(\hat{C}, \beta)$ is from $\mathcal{A}_1$ to $\mathcal{A}_2$. The operation for obtaining a clause $\psi(\hat{C}, \beta)$ from a meta-clause $\hat{C}$ is called **instantiation** by β.

6.5 Instantiation of Meta-descriptions

A meta-description M is transformed into a declarative description P_β by the instantiation of all meta-clauses in M by β in $\mathcal{B}$, i.e.,

$$P_\beta = \{C \mid C = \psi(\hat{C}, \beta) \mid \hat{C} \in M\}.$$

The operation for obtaining a declarative description P_β from a meta-description M, which is denoted by

$$P_\beta = \psi(M, \beta),$$

is also called **instantiation** by β.

6.6 Meaning of a Meta-description and Its Equivalence

Let Dom be the set of all pairs (β, Q) of elements β in $\mathcal{B}$ and elements Q in $Dscr(\mathcal{A}_1, \mathcal{A}_2)$, that is,

$$Dom = \mathcal{B} \times Dscr(\mathcal{A}_1, \mathcal{A}_2).$$

Given an arbitrary (β, Q) in Dom, a meta-description M determines a declarative description $D \cup Q \cup \psi(M, \beta)$. In this sense, a meta-description M "represents" all declarative descriptions in

$$\{P \mid P = D \cup Q \cup \psi(M, \beta), (\beta, Q) \in Dom\}.$$

Definition 1 *Let D be a set of definite clauses*

and M a meta-description. The meaning of M with respect to D is a mapping

$\overline{\mathcal{M}}_D(M) : Dom \rightarrow 2^{\mathcal{G}}$,

which maps an arbitrary element (β, Q) in Dom into

$\mathcal{M}(D \cup Q \cup \psi(M, \beta))$.

□

The "meaning" of M with respect to D includes all information of the meaning of declarative descriptions $D \cup Q \cup \psi(M, \beta)$ for all elements (β, Q) in *Dom*.

We define the equivalence between meta-descriptions.

Definition 2 *Let D be a set of definite clauses. Let M_1 and M_2 be meta-descriptions. M_1 and M_2 are* **equivalent** *with respect to D iff their meanings with respect to D are identical, that is,*

$\overline{\mathcal{M}}_D(M_1) = \overline{\mathcal{M}}_D(M_2)$. □

6.7 A Rewriting Rule determined by a Pair of Meta-descriptions

A pair of meta-descriptions determines a rewriting rule as follows.

Definition 3 *Let D be a set of definite clauses. Let M_1 and M_2 be meta-descriptions. A rewriting rule determined by M_1 and M_2 with respect to D, denoted by $rr_D(M_1, M_2)$, is the set of all pairs (P_1, P_2) of two declarative descriptions*

$P_1 = D \cup Q \cup \psi(M_1, \beta)$ *and*

$P_2 = D \cup Q \cup \psi(M_2, \beta)$

for any element (β, Q) in Dom. □

Theorem 2 *Let D be a set of definite clauses. Let M_1 and M_2 be meta-descriptions. If M_1 and M_2 are* **equivalent** *with respect to D, then the rewriting rule $rr_D(M_1, M_2)$ is an* **equivalent transformation rule.**

Proof. Assume that $(P_1, P_2) \in rr_D(M_1, M_2)$. From the definition, there exists an element (β, Q) in *Dom* such that

$P_1 = D \cup Q \cup \psi(M_1, \beta)$, and

$P_2 = D \cup Q \cup \psi(M_2, \beta)$.

Since M_1 and M_2 are equivalent with respect to D, we have

$\mathcal{M}(P_1) = \mathcal{M}(P_2)$.

Therefore, $rr_D(M_1, M_2)$ is an equivalent transformation rule. □

6.8 Meaning of Rewriting Rules based on a Meta-system

A rewriting rule

$rr_D(\{h \leftarrow \hat{A}\}, \{h \leftarrow \hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n\})$

is often denoted by

$\hat{A} \rightarrow \hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n$.

By using this rewriting rule,

$H \leftarrow \{\phi(\hat{A}, \theta)\} \cup B$

is transformed into

$H \leftarrow \{\phi(\hat{B}_1, \theta), \phi(\hat{B}_2, \theta), \cdots, \phi(\hat{B}_n, \theta)\} \cup B$

for an arbitrary $\beta = (\theta, H, B)$ in $\mathcal{B}$.

More generally, a rewriting rule with m bodies $(m = 0, 1, 2, \cdots)$

$rr_D($ { $h \leftarrow \hat{A}$},
{ $h \leftarrow Body_1$,
$h \leftarrow Body_2$,
$\cdots$
$h \leftarrow Body_m$ })

is denoted by

$\hat{A} \rightarrow Body_1$;
$\rightarrow Body_2$;
$\cdots$
$\rightarrow Body_m$.

For an arbitrary $\beta = (\theta, H, B)$ in $\mathcal{B}$, this rewriting rule transforms

$H \leftarrow \{\phi(\hat{A}, \theta)\} \cup B$

into m meta-clauses * :

$H \leftarrow \phi(Body_1, \theta) \cup B$.
$H \leftarrow \phi(Body_2, \theta) \cup B$.
$\cdots$
$H \leftarrow \phi(Body_m, \theta) \cup B$.

7 A Theory for Rule Generation

7.1 Definition of Meta-rules

A meta-rule is a rule that rewrites meta-descriptions. Repeated application of meta-

* $\phi(Body, \theta) \stackrel{\text{def}}{=} \{\phi(\hat{B}_1, \theta), \phi(\hat{B}_2, \theta), \cdots, \phi(\hat{B}_n, \theta)\}$ when $Body = \{\hat{B}_1, \hat{B}_2, \cdots, \hat{B}_n\}$

rules to meta-descriptions is called **meta-computation.**

Formally, a meta-rule mr is a set of pairs of meta-descriptions on a meta-system Δ, i.e.,

$$mr \subset MD(\Delta) \times MD(\Delta).$$

A meta-description M_1 is rewritten into a meta-description M_2 by a meta-rule mr, denoted by $M_1 \stackrel{mr}{\rightarrow} M_2$, iff $(M_1, M_2) \in mr$. Let MR be a set of meta-rules. $M_1 \stackrel{MR}{\rightarrow} M_2$ iff there is a meta-rule mr in MR such that $M_1 \stackrel{mr}{\rightarrow} M_2$. When MR is obvious from the context, it is also written as $M_1 \rightarrow M_2$.

7.2 Correctness of meta-rules

We define the correctness of meta-rules.

Definition 4 *Let D be a set of definite clauses. A meta-rule mr is* **correct** *with respect to D iff, for each rewriting pair $(M_1, M_2) \in mr$, M_1 and M_2 are equivalent with respect to D, that is,*

$$(M_1, M_2) \in mr \rightarrow \overline{\mathcal{M}}_D(M_1) = \overline{\mathcal{M}}_D(M_2).$$

7.3 Correctness of Meta-computation

Theorem 3 *Let D be a set of definite clauses and MR a set of meta-rules. Let*

$$M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow \cdots \rightarrow M_n$$

be an arbitrary rewriting sequence of meta-descriptions obtained by using meta-rules in MR. If all meta-rules in MR are **correct** *with respect to D, then a set $rr_D(M_1, M_n)$ obtained from this sequence is an* **equivalent transformation rule.**

Proof. Assume that

$$M_1 \rightarrow M_2 \rightarrow \cdots \rightarrow M_n$$

is obtained by repeated application of meta-rules in MR. Then, each adjacent pair of rewriting

$$M_i \rightarrow M_{i+1} \quad (i = 1, 2, \cdots, n-1)$$

preserves the meaning with respect to D, i.e.,

$$\overline{\mathcal{M}}_D(M_i) = \overline{\mathcal{M}}_D(M_{i+1}) \quad (i = 1, 2, \cdots, n-1).$$

Therefore,

$$\overline{\mathcal{M}}_D(M_1) = \overline{\mathcal{M}}_D(M_n).$$

Thus, from Theorem 2, a rule $rr_D(M_1, M_n)$ is an equivalent transformation rule. □

7.4 A Method for Rule Generation

Let D be a set of definite clauses. We define a method for generating ET rules, each of which has a given meta-atom in its left-hand side.

1. Assume that a meta-atom $\hat{A}$ is given.
2. Prepare a set MR of correct meta-rules with respect to D.
3. Let $\hat{C}$ be a meta-clause $h \leftarrow \hat{A}$. Let M_1 be a meta-description $\{\hat{C}\}$.
4. Transform M_1 into M_n by repeated application of meta-rules in MR, i.e.,
   $$M_1 \rightarrow M_2 \rightarrow \cdots \rightarrow M_n.$$
5. From $M_1 = \{h \leftarrow \hat{A}\}$ and
   $$M_n = \{\ h \leftarrow Body_1,\ h \leftarrow Body_2,\ \cdots\ h \leftarrow Body_m\ \},$$
   obtain a rewriting rule $rr_D(M_1, M_n)$, which is also denoted as
   $$\hat{A} \rightarrow Body_1;\ \rightarrow Body_2;\ \cdots\ \rightarrow Body_m.$$

By Theorem 3, rewriting rules obtained by this procedure are equivalent transformation rules.

8 Example in the Term Domain

8.1 A Meta-system

A meta-system for the term domain is proposed. This meta-system can be used to generate many ET rules, including some ET rules presented in this paper.

Meta-terms are of the same form as usual terms (simple terms and compound terms) in logic programming, but &-variables and #-variables are used instead of usual variables. For example, "terms" such as $f(\&A, a, \#Z)$, a, $\&B$, and $\#X$ are meta-terms. An &-variable is a variable that begins with & (such as $\&A$) and can be replaced with an arbitrary usual term. A #-variable is a variable that begins with #

(such as $\#Z$) and can be replaced with an arbitrary usual variable.

Let R_1 and R_2 be mutually disjoint subsets of R_P. We assume that R_1 and R_2 are both infinite sets. Let $\mathcal{A}_1$ be the set of all atoms, each of which consists of a predicate in R_1 and a sequence of (possibly zero) terms. Let $\mathcal{A}_2$ be the set of all atoms, each of which consists of a predicate in R_2 and a sequence of (possibly zero) terms.

Let $\hat{\mathcal{A}}_1$ be the set of all expressions, each of which consists of a predicate in R_1 and a sequence of (possibly zero) meta-terms. For instance, when *equal* and *app* are elements in R_1,

$equal(\&X, [\&A|\#Z])$

and

$app(\#Z, [a|\&Y], \#Z)$

are elements in $\hat{\mathcal{A}}_1$.

Let $\mathcal{A}_1$ be the set of all atoms consisting of a predicate in R_1 and a sequence of (possibly zero) terms.

Let Θ be the set of all mappings θ, from the set of all &-variables and all #-variables to the set of all meta-terms, that satisfy the following conditions.

1. An &-variable is mapped into a term.
2. A #-variable is mapped into an ordinary variable.

Let ϕ be a mapping from $\hat{\mathcal{A}}_1 \times \Theta$ to $\mathcal{A}_1$ such that $\phi(\hat{A}, \theta)$ is the atom obtained by substituting all &- and #- variables v in $\hat{A}$ with $\theta(v)$.

Let $\mathcal{B}$ be the set of all $\beta = (\theta, H, B)$ in $\Theta \times \mathcal{A}_2 \times \mathcal{A}_1^*$ that satisfies the following condition. Each #-variable v is mapped by θ into an ordinary variable that

(b-1) is different from variables substituted for other #-variables,

(b-2) is not included in any terms substituted for &-variables, and

(b-3) does not appear in H and B.

Then,

$\Delta = \langle \hat{\mathcal{A}}_1, \mathcal{A}_1, \Theta, \phi, \mathcal{B} \rangle$,

is obviously a meta-system.

8.2 Meta-rules

The next three rules are simple meta-rules.

(a) $initial(*A, *B) \rightarrow app(*A, \%Y, *B)$.

(b) $app(*X, *Y, *Z)$
$\rightarrow equal(*X, [\,])$,
$equal(*Y, *Z)$;
$\rightarrow equal(*X, [\%A|\%V])$,
$equal(*Z, [\%A|\%W])$,
$app(\%V, *Y, \%W)$.

(c) $app(*X, \%Y, *Z) \rightarrow initial(*X, *Z)$.

All of these transform a meta-description into another one by replacing a meta-atom (called a **target meta-atom**) in a body of a meta-clause (called a **target meta-clause**) with a sequence of meta-atoms.

In these meta-rules, we use $*$-variables and %-variables as well as constants, but not &-variables, #-variables, and ordinary variables.

$*$-variables are variables that start with $*$ (such as $*Y$), and %-variables are variables that start with % (such as $\%Y$). A $*$-variable can be replaced with any meta-term, and a %-variable with any #-variable. On substitution for these variables, the following "exclusive condition regarding %-variables" should be satisfied.

[Exclusive Condition Regarding %-Variables]
Each #-variable substituted for a %-variable does not appear in the meta-clauses before and after the transformation for a reason other than substitution for the %-variable.

8.3 Meta-rules for Equality Atoms

In addition to the meta-rules above, we introduce four meta-rules that are applied to for equality meta-atoms. $\langle true \rangle$ and $\langle false \rangle$ are used in the same manner as in Section 2.2.

(Ea) $equal([*A|*X], [*B|*Y])$
$\rightarrow equal(*A, *B)$,
$equal(*X, *Y)$.

(Eb) $equal(*X, *X) \rightarrow \langle true \rangle$.

(Ec) $equal([\,], [*A|*X]) \rightarrow \langle false \rangle$.
$equal([*A|*X], [\,]) \rightarrow \langle false \rangle$.

(Ed) When $*V$ is a #-variable, and
$*Z$ does not include the #-variable:
$equal(*V, *Z) \rightarrow \{*V/*Z\}$.
$equal(*Z, *V) \rightarrow \{*V/*Z\}$.

When there is a meta-atom of the form $equal([*A|*X],[*B|*Y])$ in the target meta-clause, this meta-atom can be rewritten, by meta-rule (Ea), into two meta-atoms of the form $equal(*A, *B)$ and $equal(*X, *Y)$. By meta-rule (Eb), an *equal* meta-atom whose two arguments are identical can be removed. By meta-rule (Ec), a target meta-clause that includes in the body an *equal* meta-atom whose arguments are a null list and a non-empty list can be deleted. Meta-rule (Ed) means that when there is an *equal* meta-atom whose arguments are a #-variable and a meta-term that does not include the #-variable, the *equal* meta-atom is removed and the #-variable is replaced with the meta-term. In this case, all occurrences of the #-variable in the body of the meta-clause should also be changed in the same way by the replacement.

8.4 Meta-computation

When a meta-atom $initial(\&X,[\,])$ is given as an input pattern, we have the following transformation. In this case, each meta-description consists of one or two meta-clauses.

(1) $h \leftarrow initial(\&X,[\,])$.
by meta-rule (a)

(2) $h \leftarrow app(\&X,\#Y,[\,])$.
by meta-rule (b)

(3) $h \leftarrow equal(\&X,[\,]),$
$equal(\#Y,[\,])$.
$h \leftarrow equal(\&X,[\#A|\#V]),$
$equal([\,],[\#A|\#W]),$
$app(\#V,\#Y,\#W)$.
by meta-rule (Ec)

(4) $h \leftarrow equal(\&X,[\,]),$
$equal(\#Y,[\,])$.
by meta-rule (Ed)

(5) $h \leftarrow equal(\&X,[\,])$.

Similarly, when a meta-atom $initial(\&X,[\&A|\&Z])$ is given as an input pattern:

(6) $h \leftarrow initial(\&X,[\&A|\&Z])$.
by meta-rule (a)

(7) $h \leftarrow app(\&X,\#Y,[\&A|\&Z])$.
by meta-rule (b)

(8) $h \leftarrow equal(\&X,[\,]),$
$equal(\#Y,[\&A|\&Z])$.
$h \leftarrow equal(\&X,[\#B|\#V]),$
$equal([\&A|\&Z],[\#B|\#W]),$
$app(\#V,\#Y,\#W)$.
by meta-rule (Ed)

(9) $h \leftarrow equal(\&X,[\,])$.
$h \leftarrow equal(\&X,[\#B|\#V]),$
$equal([\&A|\&Z],[\#B|\#W]),$
$app(\#V,\#Y,\#W)$.
by meta-rule (Ea)

(10) $h \leftarrow equal(\&X,[\,])$.
$h \leftarrow equal(\&X,[\#B|\#V]),$
$equal(\&A,\#B),$
$equal(\&Z,\#W),$
$app(\#V,\#Y,\#W)$.
by meta-rule (Ed)

(11) $h \leftarrow equal(\&X,[\,])$.
$h \leftarrow equal(\&X,[\&A|\#V]),$
$equal(\&Z,\#W),$
$app(\#V,\#Y,\#W)$.
by meta-rule (Ed)

(12) $h \leftarrow equal(\&X,[\,])$.
$h \leftarrow equal(\&X,[\&A|\#V]),$
$app(\#V,\#Y,\&Z)$.
by meta-rule (c)

(13) $h \leftarrow equal(\&X,[\,])$.
$h \leftarrow equal(\&X,[\&A|\#V]),$
$initial(\#V,\&Z)$.

Meta-rule (c) can be applied to the second meta-clause in (12) since $\#Y$ does not appear except in the second argument of $app(\#V,\#Y,\&Z)$ in (12).

8.5 Examples of Rule Generation

Two rules can be created from the example in this section. Since meta-clause (1) is transformed into (5), we obtain a rule:

(p) $initial(\&X,[]) \rightarrow equal(\&X,[])$.

By this rule, an atom that matches the meta-atom $initial(\&X,[])$ is changed into $equal(\&X,[])$.

Consider the following ET rule:

$equal(\&X,\&Y) \rightarrow \{\&X = \&Y\}$.

By the composition of the rule (p) and this rule for equality, we obtain a new ET rule (which is represented here using the notation in Section 2.5):

r7: $initial(X,[]) \rightarrow \{X = []\}$.

Furthermore, since meta-clause (6) is transformed into (13), the following rule is obtained.

(q) $initial(\&X,[\&A|\&Z])$
$\rightarrow equal(\&X,[])$
$\rightarrow equal(\&X,[\&A|\#V]),$
$initial(\#V,\&Z)$.

Finally, we have:

r8: $initial(X,[A|Z]) \rightarrow \{X = []\};$
$\rightarrow \{X = [A|V]\},$
$initial(V,Z)$.

9 Comparison

9.1 Program Transformation v.s. Rule Generation

Rule generation (RG) is different from program transformation [2] [4](PT). In logic programming, a set of definite clauses is regarded as specification and as a program at the same time. PT in logic programming transforms a set of definite clauses into a new set of definite clauses. On the other hand, RG generates ET-rules from a set of definite clauses (and a query).

9.2 Ordinary Variables v.s. Meta-variables

Transformation of meta-descriptions is similar to program transformation of logic programs. By omitting & and # from meta-variables, we have a sequence similar to a program transformation sequence [4]. For instance, from the first example in Section 8.4, we have:

(1') $h \leftarrow initial(X,[])$.
(2') $h \leftarrow app(X,Y,[])$.
(3') $h \leftarrow equal(X,[]),$
$equal(Y,[])$.
$h \leftarrow equal(X,[A|V]),$
$equal([],[A|W]),$
$app(V,Y,W)$.
(4') $h \leftarrow equal(X,[]),$
$equal(Y,[])$.
(5') $h \leftarrow equal(X,[])$.

This can also be obtained by using the following ET-rules:

$initial(X,Z) \rightarrow app(X,Y,Z)$.
$app(X,Y,Z) \rightarrow equal(X,[]),equal(Y,Z);$
$\rightarrow equal(X,[A|V]),equal(Z,[A|W]),$
$app(V,Y,W)$.
$equal(X,Y) \rightarrow \{X = Y\}$.

However, these rules can transform (5') further into a clause ($h \leftarrow$.), which results in an **incorrect** rule:

$initial(X,[]) \rightarrow \langle true \rangle$.

When these three rules are used, we can not explain why transformation should be stopped at (5'). Therefore, equivalent transformation of usual definite clauses can not be used (at least without change) for generation of ET rules.

The difference between #-variables and &-variables is essential to correct generation of ET rules. Meta-rules can prevent further transformation at (5) since there is no rule applicable to $equal(\&X,[])$. On the other hand, when given $equal(\#X,[])$ instead of $equal(\&X,[])$, meta-rule (Ed) can remove it.

10 Concluding Remarks

In order to develop a foundation for program synthesis in the ET paradigm, we have proposed a method for generating ET rules from a set of definite clauses and a meta-atom. A new notion called meta-description, which consist-

s of "meta-clauses," was introduced. A meta-description is a representative expression for many declarative descriptions. An ET rule is generated by transforming meta-descriptions using meta-rules.

One of the most important characteristics of the proposed method is modularity. A meta-rule is modular in the sense that its correctness can be judged independently of other meta-rules. Meta-rules can be applied in any order, while the folding rule in logic programming can not.

References

[1] K. Akama, Y. Shigeta, and E. Miyamoto, "Problem Solving by Equivalent Transformation of Logic Programs", 5th International Conference on Information Systems Analysis and Synthesis (ISAS'99), 1999.

[2] Futamura,Y., Nogi,K. and Takano,A.: Essence of Generalized Partial Computation, *Theoretical Computer Science*,90, pp61–79 (1991).

[3] J.W. Lloyd, Foundations of Logic Programming, Second edition, Springer-Verlag, 1987.

[4] A. Pettorossi and M. Proietti, "Transformation of Logic Programs: Foundations and Techniques", *The Journal of Logic Programming*, Vol.19/20, 1994, pp.261–320.